NASA Contractor Report 182052
ICASE Report No. 90–41

ICASE

EXECUTION TIME SUPPORT FOR ADAPTIVE SCIENTIFIC ALGORITHMS ON DISTRIBUTED MEMORY MACHINES

Harry Berryman
Joel Saltz
Jeffrey Scroggs

Contract No. NAS1–18605
May 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines*

Harry Berryman and Joel Saltz and Jeffrey Scroggs

Institute for Computer Applications in Science and Engineering,
NASA Langley Research Center,
Hampton VA 23065

Abstract

We consider optimizations that are required for efficient execution of code segments that consists of loops over distributed data structures. The PARTI execution time primitives are designed to carry out these optimizations and can be used to implement a wide range of scientific algorithms on distributed memory machines.

These primitives allow the user to control array mappings in a way that gives an appearance of shared memory. Computations can be based on a global index set. Primitives are used to carry out gather and scatter operations on distributed arrays. Communications patterns are derived at runtime, and the appropriate send and receive messages are automatically generated.

*This work is supported under NASA contract NAS1-18605, and by the U.S. Office of Naval Research under Grant N00014-86-K-0310

1 Introduction

Efficient implementation of scientific codes on distributed memory architectures requires special techniques both at run-time and at compile-time. The PARTI (Parallel Automated Runtime Toolkit at ICASE) system is a set of primitives that can be used to implement a wide range of scientific algorithms on distributed memory machines. These primitives support various run-time operations required by programs that make use of an embedded shared name space on a distributed machine. The user can carry out gather and scatter operations on distributed arrays using a global index set. Communications patterns are derived at runtime, and the appropriate send and receive messages are automatically generated.

The PARTI primitives are initialized by specifying a mapping into distributed memory for each globally defined multidimensional array. The primitives include procedures that allow one to scatter and gather array elements in the distributed memory. The PARTI tools are organized into two levels. The lower level supports memory operations such as scatters and gathers across processors. The higher level binds mapping information to distributed arrays and uses this information to call the lower level primitives. The primitives allow the storage of information about memory access patterns so that memory operations with the same address calculations need not repeat these calculations. Send/receive schedules are generated for memory operations. The schedules may be stored and reused as well. This is particularly important for the implementation of iterative algorithms.

The ideas incorporated in PARTI are specifically aimed at computations on distributed memory machines in which the structure of the computation depends on the input data. Run-time support must be incorporated as part of the distributed implementation of such computations. Directly incorporating the primitives into applications programs allow investigation of the usefulness and relevance of various optimizations. In this paper we demonstrate and benchmark these primitives using two different programs. The first is an adaptive method for solving partial differential equations, the second is a kernel from an unstructured mesh code.

1.1 Related Research

Williams [16] describes a programming environment for calculations with unstructured triangular meshes using distributed memory machines. In [16], collections of distributed array accesses are translated into an efficient set of inter-node messages. Similar mechanisms for translating an irregular pattern of array accesses into inter-node messages have been proposed in order to make it possible to efficiently distribute loops where some array references are made through a level of indirection. Work on this topic was presented by the present authors in [12],[8], [9] as well as by Mehrotra and Van Rosendale [7, 6].

In this paper, we present primitives that can be used directly by programmers that allow users to carry out gather and scatter operations over global index sets on distributed arrays.

Callahan and Kennedy [3], Rogers and Pingali [10], and Rosing and Schnabel [11] suggest execution time resolution of communications on distributed machines. None

of these utilize information on repeated patterns of communications. The Linda system[1] provides an associative addressing scheme by which a reference to variables can be resolved at execution time. This in essence provides a shared name space for distributed memory machines; however, the shared name space does not allow users to determine how data is to be partitioned between processors. The costs associated with this lack of data locality can be extremely high in some cases [12].

2 PARTI Primitives

The PARTI primitives currently consist of two levels. The most fundamental of the primitives are the Level 0 Primitives. They consist of routines to *gather* and/or *scatter* (read and write) values to elements of one dimensional arrays $\texttt{aloc}^j$ defined on each processor j. Each $\texttt{aloc}^j$ is local to processor j; it is not viewed as a distributed array by the Level 0 Primitives.

2.1 Level 0 Gather and Scatter

Level 0 gathers and scatters are accomplished by using three routines: *Scheduler*, *Gather Exchanger*, and *Scatter Exchanger*.

Scheduler on processor P^i is passed a list of indices K_j into each $aloc^j$ from which data is to be fetched and produces a schedule S that is used by both exchangers.

On processor P^i, Gather Exchanger inputs

1. a buffer into which the fetched elements are to be placed
2. the location of array $aloc^i$
3. the schedule S produced by Scheduler

Gather Exchanger executes sends and receives that fetch from each processor P^j the appropriate elements from the array $aloc^j$. Then it places these elements into the user-supplied buffer.

Scatter Exchanger is passed

1. a buffer from which each scattered datum is to be obtained
2. the location of array $aloc^i$
3. the schedule S produced by Scheduler

Scatter Exchanger executes sends and receives that put on each processor P^j the appropriate elements from the buffer. Then Scatter Exchanger places these elements into the appropriate elements of array $aloc^j$.

2.1.1 Functioning of the Scheduler and Exchangers

Exchange procedures for both the scatter and the gather have three stages. They permute data into buffers to be sent. They carry out the needed communication, then they perform another permutation.

The scheduler first determines how many messages each processor must send and receive during the data exchange phase. Defined on each processor P^i is an array $\texttt{nmsgs}^i$. Each processor sets its value of $\texttt{nmsgs}^i(j)$ to 1 if it needs data from processor j or to 0 if it does not. The scheduler then replaces nmsgs with the element-by-element sum $\texttt{nmsgs}^i(j) \leftarrow \sum_k nmsgs^k(j)$. This operation utilizes a function that imposes a fan-in tree to find the sums. Since the resulting sum is kept in $\texttt{nmsgs}^i$, at the end of the fan-in on every processor, $\texttt{nmsgs}^i(j)$ is the number of messages that processor P^i must send during the exchange phase. Next, each processor sends a *request list* to every other processor. The request list sent from processor P^p to processor P^q contains the indices of data needed by processor P^p that are stored on processor P^q.

The number of non-empty request lists each processor will receive is equal to the number of messages that the processor will send in the exchange phase. Each request list is placed in an array indexed by the processor from which the list came. When the scheduler is finished, each processor has an array of request lists obtained from other processors. The j^{th} element of this array contains the request list obtained from processor j. At this point in the execution, each processor P^i knows which elements of $aloc^i$ must be sent to other processors. This information is used to generate the schedule S of pairs of send and receive statements. These send/receive pairs will exchange the requested data for either a gather or a scatter. Thus, both the gather and the scatter call the *exchanger* routine. The exchanger is passed the schedule S with the required buffer space. It then carries out the required communication.

2.1.2 Additional Exchangers

In addition to the Level 0 exchanger, we have found it useful to develop hybrids of the gather and scatter that perform remote operations on distributed array data. For example, the scatter_add adds data elements $\texttt{D}_1, ..., \texttt{D}_{n_j}$ to elements $\texttt{aloc}^j(\texttt{k}_1)$,.., $\texttt{aloc}^j(\texttt{k}_{n_j})$. Similar exchanges perform distributed subtractions, multiplications and divisions.

2.2 Level 1 Primitives

The Level 1 Primitives are the user interface between an application code and the Level 0 Primitives. Use of Level 1 Primitives allows the dynamic allocation of distributed multi-dimensional arrays and supports data transfer between these arrays. The Level 1 Primitives consist of declaration procedures and of communication procedures. The intermediate level PARTI declaration procedures allow the user to declare a dynamically allocated distributed array in a way that allows specification of how the array is to be partitioned between processors. Coupled to these distributed array declarations are the intermediate level gather and scatter procedures. These procedures are designed to allow users to fetch or store array elements from the

distributed memory in a way that does not require the user to keep track of where array elements are stored. This makes it relatively straightforward to write codes that allow data structures to be repartitioned during program execution. All memory is allocated or declared in the programs that call PARTI procedures; memory locations are passed to the procedures that perform array initializations. Users write programs that contain a combination of

1. code written to execute on individual processors
2. communications calls that consist of gathers or scatters to distributed arrays
3. communication calls that consist of zero-level gathers or scatters
4. send and receive message passing calls

2.2.1 Level 1 Declaration Procedures

The declaration procedures in the PARTI primitives allow the user to describe how a data array is mapped into the distributed memory of the machine. This is accomplished by specifying the mapping of the data to a virtual processor array, then describing the relationship between the virtual processor array and the original processor array.

Specified in an array initialization is a processor grid G of arbitrary dimension and size. This processor grid is automatically gray coded in the current implementation. Embedded in G is an array of virtual processors V. The embedding is specified by the user. This two-stage specification of processor sets allows embedding different distributed arrays into different subarrays V of G. Note also that different distributed arrays can be initialized with different processor grids G or with the same G but with different virtual processor subarrays V.

Data arrays are mapped onto virtual processor subarrays in any one of a number of ways. We support tensor product mappings in which each array dimension is partitioned independently or is left undistributed. Following [7], we support *blocked* partitionings where each processor receives an equal number of contiguous array elements along a given dimension, and *cyclic* distributions in which elements are distributed stripped fashion across the processors. We also support *enumerated* distributions in which mappings are supported by distributed translation tables. The number of dimensions of an array to be distributed must match the dimensionality of G.

2.2.2 Level 1 Communication Procedures

The Level 1 `Gather exchanger` and `Scatter exchanger` routines allow communication of user data based on the global index set. The Level 1 Scatter Exchanger inputs lists of distributed array indices and values. It places the values in the distributed memory locations specified by the indices (and the initially supplied array mapping). The intermediate level Gather Exchanger inputs lists of distributed array indices along with a pointer to a memory buffer in the calling processor. Data values from the appropriate distributed memory locations are obtained and placed in the

calling processor's buffer. An initialization or Scheduler procedure call is required for Gather exchanger or Scatter exchanger. The initialization procedure precomputes the locations of the data that will to be sent and received by each processor. This initialization is needed only once–it may be reused any number of times. The initialization will be described first.

The initialization combines the mapping information provided by the declaration routines with a specific list of global indices. The result is a set of communication calls coupled with some pre- or post-movement of the data (for the gather and scatter, respectively). The communication plus movement is executed when the gather or scatter is called.

Each array A distributed with the Level 1 Primitives is treated by the Level 0 routines as a set of *aloc* arrays defined on each processor. Elements of A are gathered by calculating corresponding indices of *aloc* arrays and then executing a Level 0 gather.

The index of a D-dimensional array is translated to a physical processor p and an index i to *aloc*. Index i is generated from the user-specified index by using the mapping from the data to the virtual processor array. The physical processor P is determined by locating the virtual processor in V, then using the embedding information to obtain the position in G. The physical processor corresponding to the calculated position in G can be obtained from the gray code. Notice that the physical memory locations of the various *aloc*s are not used directly in the calculations. Rather, the index i is passed to processor P. This has the advantage of allowing each processor to store its data in different physical memory locations.

The steps involved in performing a gather on an index list L into A are outlined here. Translations described above produce a list of indices L_i and a list of processors L_p that correspond to L. These lists are used to determine the scheduling of a low level gather and are sorted by processor. The sort results in a permutation array L_{PERM} that reorders the elements of L. Since moving and/or copying of data is avoided, the efficiency of the gather is increased. A Level 1 gather is performed from executing the scheduled zero-level gather and then using L_{PERM} to reorder the values obtained by each processor. In many scientific programs it is also useful to execute isomorphic gathers on different arrays. Recall that actual memory addresses are only bound to the intermediate and the zero-level primitives after all of the optimizations are carried out. Consequently, the same zero-level gather schedule and the same permutation array L_{PERM} can be associated with a number of different distributed arrays. Setup costs are amortized when the same pattern of communication is carried out many times.

The Level 1 Gather Exchanger and Scatter Exchanger procedures are designed to allow users to fetch or store array elements from the distributed memory in a way that does not require the user to keep track of where array elements are stored. It is relatively straightforward to write codes that allow data structures to be repartitioned during program execution. Declaration procedures can be called to repartition the distributed data, and the application can be written in a partitioning-independent manner.

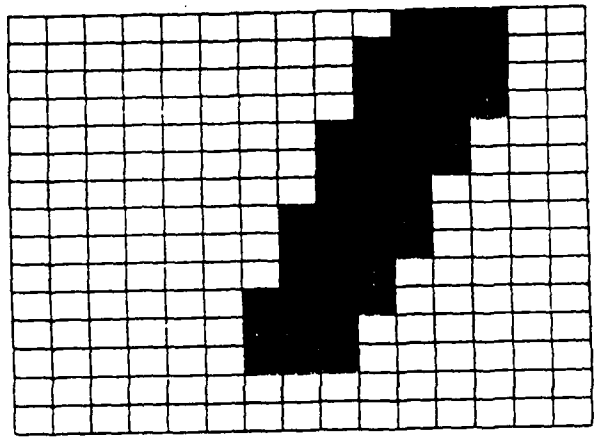

Figure 1: Two-mesh refinement.

3 The Use of PARTI Primitives in Adaptive and Unstructured Applications

We present one complete code and one computational kernel to motivate the discussion of our execution time optimizations and the PARTI (Parallel Automated Runtime Toolkit at ICASE) primitives.

3.1 Adaptive Mesh Partial Differential Equation Solver

The first example is a technique for adaptive refinement. The method will be described in the context of the solution to

$$u_t + f(u)_x + g(u)_y - \epsilon \Delta u = 0$$

in the presence of a shock where the profile (detailed shape) of the shock is desired. For the method discussed here, resolution of the profile implies that a highly refined grid must be used in a neighborhood of the shock. The theory behind the algorithm has been described [4] so only an algorithmic description of the method will be presented here. In this algorithm, the structure of the computations changes with time and a non-uniform communication pattern arises due to the sharing of data between grids.

The method initially computes the solution on a coarse mesh. An error estimator is then applied to determine the regions that will be covered by a refined mesh. An example mesh from this two-level refinement is shown in Figure 3.1.

The solution is time-dependent. Time-marching on the refined mesh is performed by taking many (e.g. 100) time steps on the refined mesh for a single coarse-grid time step. Let U represent the solution, k be the temporal step-counter, and (i, j) represent the discrete location on a spatial grid. The subscripts c and r are used to refer to the coarse and refined meshes, respectively. The data structure used for the coarse mesh is a two-dimensional array. The solution on the refined mesh is represented by a three-dimensional data structure in which the third index rep[illegible]

a block of the refined mesh (each block corresponds to a single coarse grid square), and first two indices represent the spatial location within the block. The general structure of the kernel is outlined in Algorithm 1. In general, a shock moves and

For $k_c = 1$ to K
  I. Sweep over the coarse mesh
    A. Compute U_c.
    B. Flag region that should be refined.
  II. If flagged region is not empty.
    A. Modify shape of refined region
    B. Interpolate boundary values for U_r from U_c.
    C. For $k_r = 1$ to K_r
      1. Sweep over the refined mesh
      2. Share values between blocks in U_r
    D. Inject values of refined region into coarse grid

ALGORITHM 1 *Two-mesh algorithm.*

changes shape. Thus, the refined mesh will be dynamic – its location, shape, and size all change. This means that both the communication pattern within a distributed mesh and the relationship of the two meshes will change during the execution of the program.

Classes of inter-processor communication needed to implement Algorithm 1 in a distributed computing environment are;

1. communication involved in coarse mesh sweeps (Step I.A.),
2. communication involved in fine mesh sweeps (Step II.C.1.),
3. sharing of values between the coarse and fine meshes(Steps II.B. and II.D.), and
4. communication required to modify the shape of the refined region (Step II.A.).

In our implementation of Algorithm 1, the PARTI primitives are used in all but the last set of communications.

3.2 Unstructured Mesh Kernel

Figure 2 depicts a schematic outline of a kernel from a fluid dynamics simulation. In Section 4.2 we will present experimental results obtained from a similar but slightly more complex kernel. The kernel is based on an algorithm which fills a computational domain with irregular polygons. Heuristics designed to ensure that the governing partial differential equation is solved with an approximately equal accuracy throughout the computational domain determine the area and shape of the polygons. The data structures used in solving the problem represent a bidirectional graph where vertices

For i=1 to Number-Edges

I. v_A = yold(node(i,1))
   v_B = yold(node(i,2))

II. Calculate flux using v_A, v_B.
   This calculation also uses edgedata(i,1),.., edgedata(i,SMALL)

III. y(node(i,1)) = y(node(i,1)) + flux
   y(node(i,2)) = y(node(i,2)) - flux

Figure 2: Outline of Computational Fluid Dynamics Unstructured Mesh Kernel

represent polygons and edges represent adjacency of the polygons. Sweeps over the polygons are accomplished by traversing the edges of this graph. In these codes we solve for equation values at graph vertices.

The kernel outlined in Figure 2 computes the flux across each graph edge. The indices of the two vertices connected by the `j`th graph edge are denoted by `node(j,1)` and `node(j,2)` in Figure 2. The computation of the flux terms requires `yold(node(i,1))` and `yold(node(i,2))` (step I in Figure 2). The computation also requires other information concerning graph edge `i` (step II in Figure 2). In this example, we assume that data pertaining to edge `i` is placed in row `i` of `edgedata`. In step III, flux is added to `y(node(i,1))` and flux is subtracted from `y(node(i,2))`

It is necessary to decide how the parallel loop iterations for index `i` are to be partitioned between processors. We must also specify how the data in arrays `y` and `yold` should be partitioned. No matter how we partition loop iterations and data, the structure of the problem requires that we access off-processor elements of `y` and `yold`. On the other hand, `edgedata(i,j)` and `node(i,j)` (`j = 1,2`) are used only in the `i`th parallel loop iteration so these arrays can be partitioned so that only local array accesses are needed.

In this kernel, the dependencies between elements of arrays `y` and `yold` are determined by integer array `node`. We therefore cannot accurately predict what data must be prefetched until the program executes. On distributed machines, it is typically very inefficient to fetch individual off-processor data as a need for these elements is encountered because of high communications latencies. We will use the level 0 primitives to allow us to preschedule the communications needed to efficiently prefetch off-processor data.

3.3 The Preprocessing Phase

In Figure 3, we outline the preprocessing needed to distribute the computation depicted in Figure 2. In this example, assume that `y` and `yold` are mapped in an identical manner. We also assume that *Local(x)* and *Processor(x)* are functions that return the processor and local index associated with the distributed array element

I. For all edges i assigned to processor P

if Processor(node(i,1)) $\neq$ P

concatenate Processor(node(i,1)) to $Proc_A$
concatenate Local(node(i,1)) to $Local_A$

if Processor(node(i,2)) $\neq$ P

concatenate Processor(node(i,1)) to $Proc_B$
concatenate Local(node(i,1)) to $Local_B$

II. Call Level 0 Scheduler with $Local_A$ and $Proc_A$ (schedule S_A)

Call Level 0 Scheduler with $Local_B$ and $Proc_B$ (schedule S_B)

Figure 3: Preprocessing Unstructured CFD Kernel

x. In this example, we use only the level 0 primitives.

In the first step (step I in the figure) we sweep through the edges assigned to processor P and generate lists of off-processor references into distributed arrays y and yold. On processor P, the arrays $Proc_A$ and $Local_A$ are used to store the processor and local array index that corresponds to each off-processor reference made by P to y(node(i,1)) and yold(node(i,1)). Arrays $Proc_B$ and $Local_B$ are used to store off-processor references made by P to y(node(i,2)) and yold(node(i,2)). In step II. in Figure 3, the level 0 scheduler is called with the lists of processors and local indices associated with the first vertex ($Proc_A$ and $Local_A$). The level 0 scheduler is called again with the analogous lists associated with the second vertex. The two level 0 scheduler calls produce schedules S_A and S_B respectively.

Once the schedules S_A and S_B have been obtained, we can begin the sweep over the graph edges. The schedules obtained above can be reused as long as the assignment of edges to processors is not altered and values assigned to the integer array node are not changed. In step I in Figure 4, copies of data from off-processor elements of array yold are obtained and put into arrays $\texttt{read} - \texttt{buffer}_A$ and $\texttt{read} - \texttt{buffer}_B$. In step II in this figure, we loop over all edges assigned to processor P. In step IIA, when yold(node(i,1)) is assigned to P, we can assign yold(Local(node(i,1))) to v_A. When yold(node(i,1)) is stored off-processor, v_A must be obtained from the buffer $\texttt{read} - \texttt{buffer}_A$. The analogous conditional assignment is carried out for v_B in step IIB. In step IIC the flux term is computed using v_A and v_B and edgedata(i,1),...,edgedata(i,SMALL). In step IID operations on elements of y local to processor P are preformed. Operations on non-local elements of y are deferred. For non-local elements of y assignments are made to buffers $\texttt{write} - \texttt{buffer}_A$ and to $\texttt{write} - \texttt{buffer}_B$. Finally in step III, the calculated flux elements are added and subtracted from the appropriate off-processor array elements through the use of the scatter_add and scatter_subtract exchangers. Note that we can again use schedules S_A and S_B.

I. Call Level 0 Gather Exchanger using (schedule S_A),
place data in $\texttt{read} - \texttt{buffer}_A$

Call Level 0 Gather Exchanger using (schedule S_B),
place data in $\texttt{read} - \texttt{buffer}_B$

II. For all edges i assigned to processor P

A. if Processor(node(i,1)) = P
v_A = yold(Local(node(i,1))
otherwise
$v_A = \texttt{read} - \texttt{buffer}_A(\text{Acount})$
Acount = Acount + 1

B. if Processor(node(i,2)) = P
v_B = yold(Local(node(i,2))
otherwise
$v_B = \texttt{read} - \texttt{buffer}_B(\text{Bcount})$
Bcount = Bcount + 1

C. Calculate flux using v_A, v_B.
This calculation also uses edgedata(i,1),.., edgedata(i,SMALL)

D. if Processor(node(i,1)) = P
y(Local(node(i,1)) = y(Local(node(i,1)) + flux
otherwise
$\texttt{write} - \texttt{buffer}_A(\text{Lcount}) = \text{flux}$

if Processor(node(i,2)) = P
y(Local(node(i,2)) = y(Local(node(i,2)) + flux
otherwise
$\texttt{write} - \texttt{buffer}_B(\text{Bcount}) = \text{flux}$

III. scatter_add exchanger called using schedule S_A and $\texttt{write} - \texttt{buffer}_A$

scatter_subtract exchanger called using schedule S_B and $\texttt{write} - \texttt{buffer}_B$

Figure 4: Unstructured Mesh CFD Kernel

4 Experiments

The experiments described in this paper used either a 32 processor iPSC/860 machine located at ICASE at NASA Langley research center or a 128 processor iPSC/860 machine located at Oak Ridge National Laboratories. Each processor had 8 megabytes of memory. We used the Greenhill 1.8.5 Beta version C compiler and the Greenhill 1.8.5 Beta version 4.1 Fortran compiler to generate code for the 80860 processors.

4.1 Primitives Benchmark Timings

We first measure the time required to carry out level 0 and level 1 Scheduler, Gather Exchanger and Scatter Exchanger procedure calls. We use the level 1 initialization primitive to declare a 128 by 128 element distributed array of single precision numbers. We allocate four processors configured in a 2 by 2 grid G and allocate an array block to each processor.

We use the level 1 primitives to repeatedly exchange information between two processors in the grid. We first scatter and then gather lists of array elements. In this experiment, we chose array elements in n by n sub-blocks between the upper left hand corner and the lower left hand corner of G. In performing this experiment, we measure the time required to carry out the following procedure calls:

1. Level 0 Scheduler
2. Level 0 Scatter
3. Level 0 Gather
4. Level 1 Scheduler
5. Level 1 Scatter
6. Level 1 Gather
7. iPSC/860 supplied send and receive pairs that exchange $4n^2$ bytes of data

In Table 1 we depict the results of these experiments. We present the time (in milliseconds) required to carry out the requisite data exchange using send and receive messages. We then present the ratio between the time taken by PARTI primitive calls and the time taken by the equivalent send and receive calls. Table 1 only presents Gather Exchange and Scheduler calls. The Level 0 and Level 1 Scatter Exchange calls were also timed, the results for each Scatter Exchange call were virtually identical to that of the corresponding Gather Exchange call.

We first note that for relatively large amounts of data, a Level 0 Gather Exchange takes a factor of 1.2 more time than the corresponding send/receive pair. A Level 1 Gather Exchange takes a factor of 1.6 more time than the corresponding send/receive pair. Again, for relatively large amounts of data, it costs about as much to schedule a message using the Level 0 primitives as it takes to send the message using send and receive messages. In contrast, the Level 1 Scheduler is an order of magnitude more expensive than the corresponding send/receive pairs. This relatively high cost

Table 1: Overheads for Level 0 and Level 1 Primitives

Number of Data Elements	Send Receive Time	Level 0 Gather (ratio)	Level 1 Gather (ratio)	Level 0 Scheduler (ratio)	Level 1 Scheduler ratio
100			1.2	2.1	7.0
400			1.3	1.4	9.2
900			1.5	1.3	10.7
1600			1.6	1.3	11.2
2500			1.6	1.1	11.1
3600		.2	1.6	1.0	11.2

is caused by the integer operations needed to identify each reference to a distributed two dimensional array with a processor and local storage location.

4.2 Kernel from Unstructured Mesh Code

We used an unstructured mesh that was generated to carry out an aerodynamic simulation involving a multielement airfoil in a landing configuration [5]. The unstructured mesh consists of a highly non-uniform scattering of mesh points joined together by line segments to form a set of triangular elements. The algorithm used is the Delaunay Triangulation algorithm [13]. Details of this mesh generation process can be found in [5]. The mesh used in this problem had 11143 vertices, and is shown in Figure 5. The computational algorithm we study computes convective fluxes using a method based on Roe's approximate Riemann solver [14], [15].

The computational kernel used in this experiment is very closely related to the kernel described in Figure 2. In place of each of the arrays y and yold found in the kernel described in Section 3.2, we have four different arrays y_ρ, y_u, y_v, y_p and $yold_\rho, yold_u, yold_v, yold_p$, respectively. These represent the density, pressure, and velocity components of the solution. Each call to an exchanger in Figure 4 is consequently replaced by four exchanger calls. Since the data access patterns for each of these four arrays are identical, we still need only call the scheduler twice, once with $Proc_A$ and $Local_A$ and once with $Proc_B$ and $Local_B$.

The original program was written in Fortran, we initially extracted the computational kernel and produced a sequential C version of the kernel. The multiprocessor codes developed were also written in C. When partitioning this kernel, we made only modest efforts to reduce volume of needed communication and to balance load. When we employ P processors, we partition the problem domain into P strips. All the vertices V_S in the Sth strip are assigned to a single processor. The strips are chosen to evenly partition the sum of the number of edges associated with each V_S. In order to allow us to use the Level 0 primitives, we renumbered the mesh points so that contiguously numbered mesh points are assigned to a processor. If both vertices comprising an edge were assigned to a processor P, that edge was also assigned to P. If the vertices v_1 and v_2 comprising an edge were assigned to two different processors P and Q respectively, we assigned the edge to processor Q.

We measured the time required for an iPSC/860 multiprocessor to compute the

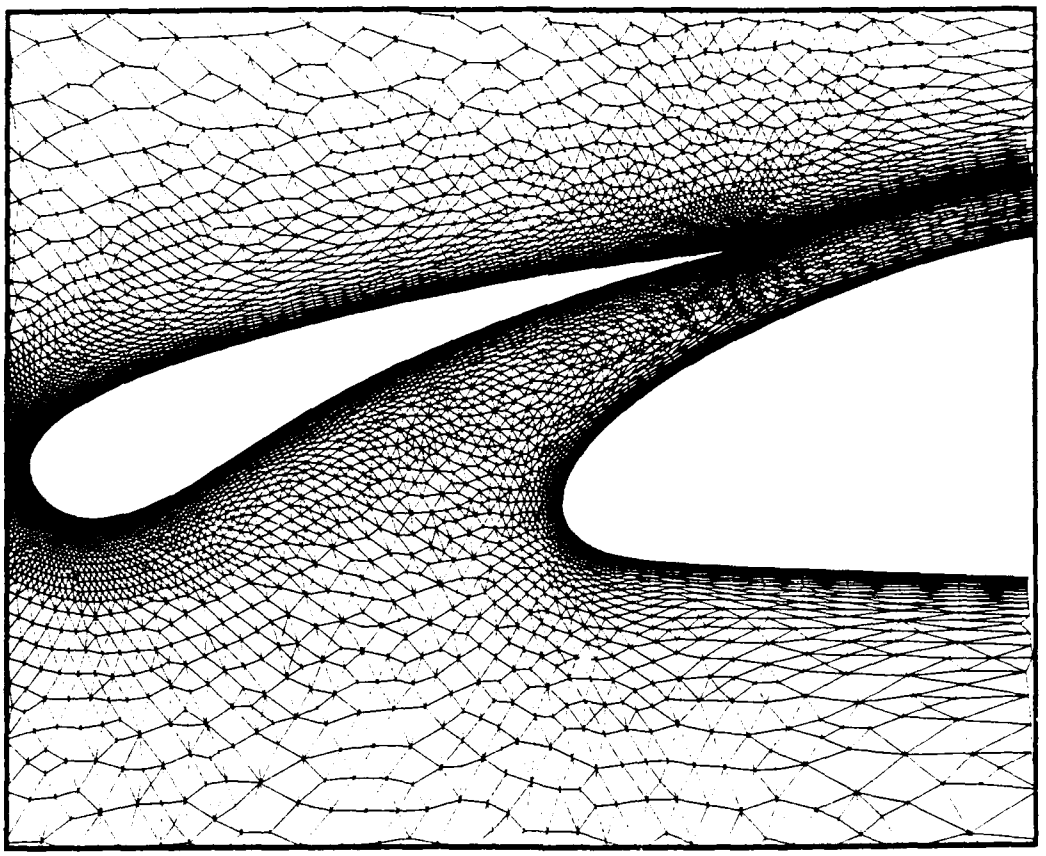

Figure 5: Unstructured Mesh for Multielement Airfoil

parallelized computational kernel as well as the time needed to carry out the preprocessing and communications steps outline in Section 3.2.

In Table 2 we first report the computational rate in Mflops for the parallelized code on 2 through 64 processors, along with a separate sequential version of the kernel timed on a single iPSC/860 node. The computational speed ranged from 3.2 Mflops for a single node to 87.3 Mflops on 64 nodes. We next depict the *total* time required to generate lists of off-processor references and to carry out the Level 0 scheduler calls (Steps I and II, Figure 3). The cost of finding the processor number and the local index number associated with the vertices in each edge took a substantial portion of this preprocessing time. For two processors the total preprocessing was 198 milliseconds. Of this total preprocessing time, 181 milliseconds was required for translating vertices to processors and local indices, and only 17 milliseconds was needed both to form lists of off-processor references and to call Level 0 schedulers (refer to Table 2). When 64 processors were used, total preprocessing required 23 milliseconds, of which 16 milliseconds was required for forming the lists of off-processor references and for calling the Level 0 schedulers.

As we mentioned in Section 3.2, the preprocessing only needs to be carried out once and can be amortized over many unstructured mesh flux calculations. The time required to carry out the flux calculations (Figure 4), is given as *total kernel time* in Table 2. Note that total kernel time *does not include the preprocessing cost.* The time required for preprocessing ranges from 21 % of the time needed to compute the computational kernel when 2 processors were used to 35 % of the cost when we used 64 processors. Finally, we report the amount of time taken by the Level 0 exchangers (Steps I and III in Figure 4. Communication costs ranged from 2% to 61% of the total cost needed for the kernel computation for 2 and 64 processors respectively. The time needed for communication in the kernel increases with the number of processors, the communication time reaches a plateau of approximately 40 milliseconds for 32 and 64 processors. This communication time could probably be significantly reduced were we to partition the domain in a way that required a smaller volume of communication. While the Level 1 primitives embed data arrays into processor arrays using a gray code, the Level 0 primitives do not provide support for such an embedding. The increase in communication time from 2 to 32 processors is largely due to the non-local communication pattern [2]; we would expect that gray coding would improve performance.

We can define parallel efficiency for a given number of processors P as the sequential time divided by the product of the execution time on P processors times P. In Table 3 we depict under the heading of *single sweep efficiency*, the parallel efficiencies we would obtain were we required to preprocess the kernel each time we carried out calculations. In reality, preprocessing time can be amortized over multiple mesh sweeps. If we neglect the time required to preprocess the problem in computing parallel efficiencies, we obtain the second set of parallel efficiency measurements presented in Table 3. The amortized parallel efficiencies we obtain range from 99.7 for 2 processors to 44.4 for 64 processors.

The processor architecture of the Intel-2/860 includes a 8K byte data cache. We can anticipate that for a fixed sized problem, the rate of computation on each node will tend to increase with concurrency. We quantified the performance effects

Table 2: Timings for Roe's Approximate Riemann Solver of Unstructured Mesh (in microsecs)

Number of Processors	Mflops	Total Preprocessing	Form off-processor Lists, Schedule	Total Kernel Time	Communication in Kernel
	Time(ms)		Time (ms)	Time(ms)	Time(ms)
1	3.1	-	-	1845	-
2	6.1	198	17	925	17
4	11.8	105	13	482	27
8	22.0	56	9	258	31
16	38.9	32	8	146	36
32	63.1	25	12	90	41
64	87.3	23	16	65	40

Table 3: Parallel Efficiencies for Roe's Approximate Riemann Solver of Unstructured Mesh

Number of	Single Sweep	Amortized	Single Processor
Processors	Efficiency	Efficiency	Speed (Mflops)
1	100.0	100.0	3.08
2	82.1	99.7	3.17
4	78.6	95.6	3.17
8	73.4	89.3	3.18
16	64.8	80.0	3.20
32	50.1	64.1	3.23
64	32.8	44.4	3.24

of the data cache by employing the *sequential program* to sweep over the edges that we assigned to processor 0 when we partitioned the problem between various numbers of processors. In Table 3, we depict the results we obtained. The rate of computation was 3.08 Mflops when we calculated the flux for the entire problem. The computational rate increased monotonically as the size of the sub-problem assigned to processor 0 decreased, reaching 3.24 Mflops in the 64 processor case.

4.3 Computational Results for Domain Decomposition Algorithm

We present computational results for the parallelized domain decomposition algorithm. This domain decomposition algorithm decides when to refine the coarse mesh using an error estimator based on the first and second derivatives of the solution [4]. This program was written in Fortran.

Table 4 depicts the total computation time in seconds required to run the example problem with the tolerance (TOL) for the second derivative set to 21. The computations were carried out for a total of 440 coarse-grid time steps. Due to memory constraints, we we unable to run this problem on fewer than eight processors. An conservative lower bound estimate of the sequential time that would be

Table 4: Adaptive Mesh Solver- Timings (seconds)

Number of Processors	Total Time	Total Scheduling Time	Gather/Scatter Time
	Time(ms)	(seconds)	seconds
1	5632*		
8	794	4	65
16	417	6	40
32	248	10	27

required to solve this problem was obtained by generating S, the sum of the time all processors spent in sweeps over the fine mesh. S includes no communication or primitive calls and the code executed has the same number of operations as would a sequential sweep over a fine mesh. The 8 processor timing took 3.2 times as long as the 4 processor timing; the ratio between the 32 processor timing and S was 22.7.

We also measure the total time required by all Level 1 schedule procedure calls. The scheduling took very little time; the overhead for scheduling ranged from 0.5 % of the total time (on 8 processors) to 4.0 % of the total time (on 32 processors). This increase in the cost of the Scheduler with increasing numbers of processors can be explained by the Scheduler's global communications phase discussed in Section 2.1.1. Finally we measured the time required for carrying out gather/scatter procedure calls. The time required for the gather/scatter procedure calls ranged from 8 % to 11 % of the execution time.

In Table 5 we depict the results obtained from running the example problem on 32 processors with a range of second derivative tolerance (TOL) values; the amount of refinement increases as TOL decreased. These problems were all continued for a total of 440 coarse-grid time steps. In Table 5 we see that for problems in which there was less refinement, the Level 1 scheduler required a larger proportion of the total time. The ratio of computation to communication time did not change significantly with the amount of refinement.

The refined mesh blocks were distributed among processors in a round robin fashion. We consequently expect the ratio of computations to data elements communicated to remain roughly constant. As the number of refined blocks increases, we expect that each processor will have to communicate with increasing numbers of other processors. A more sophisticated strategy for assigning refined blocks to processors would be likely to result in lower communication times both by reducing the volume and increasing the locality of communications.

The Level 1 Scheduler required only 4 % of the total time when TOL was equal to 21 but the Level 1 scheduler required 10 % of the total time when TOL was equal to 27. As stated above, the scheduler has a global communications phase whose cost does not depend on the amount of data to be communicated by each processor. The Estimated Optimal Computation time depicted in Table 5 is S divided by the number of processors. This gives a rough estimate of the time that would be required to solve this problem were we to have attained linear speedup.

Table 5: Adaptive Solver- Varying Tolerance (32 processors)

Tolerence	Total Time	Est. Optimal Comp Time (seconds)	Scheduling Time (seconds)	Communication Time (seconds)
21	248	176	10	27
23	186	130	9	19
25	130	91	7	15
27	92	57	6	9

5 Conclusion

The ideas incorporated in PARTI are specifically aimed at computations on distributed memory machines in which the structure of the computation depends on the input data. Run-time support must be incorporated as part of the distributed implementation of such computations.

The PARTI primitives allow the user to describe how a data array is mapped into the distributed memory of the machine. This is accomplished by specifying the mapping of the data to a virtual processor array, then describing the relationship between the virtual processor array and the original processor array. These primitives carry out scheduling operations that make it possible to efficiently carry out gather and scatter operations on distributed arrays using global indices. To illustrate the performance tradeoffs encountered in carrying out these optimizations, we presented benchmark results from an unstructured mesh kernel, and an adaptive partial differential equation solver. Our results suggest that these primitives carry out needed optimizations at a relatively low cost.

6 Acknowledgements

We would like to thank Tom Zang and David Whitaker for providing us with the computational kernel from their approximate Riemann solver [14], [15] and to Oak Ridge National Laboratory for giving us access to their iPSC/860 hypercube.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter, *Linda and friends*, IEEE Computer, (1986), pp. 26–34.

[2] S. Bokhari, *Communication overhead on the intel ipsc-860 hypercube*, Report 90-10, ICASE Interim Report, 1990.

[3] D. Callahan and K. Kennedy, *Compiling programs for distributed-memory multiprocessors*, Journal of Supercomputing, 2 (1988), pp. 151–169.

[4] J. S. Scroggs, *A physically motivated domain decomposition for singularly perturbed equations*, SIAM Journal on Numerical Analysis, (to appear, 1990).

[5] D. J. Mavriplis, *Multigrid solution of the two-dimensional Euler equations on unstructured triangular meshes*, AIAA Journal, 26 (1988), pp. 824–831.

[6] P. Mehrotra and J. Van Rosendale, *Compiling high level constructs to distributed memory architectures*, in To appear in: Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications, March 1989.

[7] ——, *Parallel language constructs for tensor product computations on loosely coupled architectures*, in Proceedings Supercomputing '89, November 1989, pp. 616–626.

[8] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley, *Principles of runtime support for parallel processors*, in Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France, July 1988, pp. 140–152.

[9] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman, *A scheme for supporting automatic data migration on multicomputers*, in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C., 1990.

[10] A. Rogers and K. Pingali, *Process decomposition through locality of reference*, in Conference on Programming Language Design and Implementation, ACM SIGPLAN, June 1989, pp. 69–80.

[11] M. Rosing and R. Schnabel, *An overview of Dino - a new language for numerical computation on distributed memory multiprocessors*, Tech. Rep. CU-CS-385-88, University of Colorado, Boulder, 1988.

[12] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, *Run-time scheduling and execution of loops on message passing machines, (to appear in Journal Parallel and Distributed Computing, April 1990)*, Report 89-7, ICASE, January 1989.

[13] N. P. Weatherill, *The generation of unstructured grids using dirichlet tessalations*, Report MAE 1715, Princeton, July 1985.

[14] D. L. Whitaker and B. Grossman, *Two-dimensional euler computations on a triangular mesh using an upwind, finite-volume scheme*, in Proceedings AIAA 27th Aerospace Sciences Meeting, Reno, Nevada, January 1989.

[15] D. L. Whitaker, D. C. Slack, and R. W. Walters, *Solution algorithms for the two-dimensional euler equations on unstructured meshes*, in Proceedings AIAA 28th Aerospace Sciences Meeting, Reno, Nevada, January 1990.

[16] R. D. Williams and R. Glowinski, *Distributed irregular finite elements*, Tech. Rep. C3P 715, Caltech Concurrent Computation Program, February 1989.

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-182052 ICASE Report No. 90-41		

4. Title and Subtitle	5. Report Date
EXECUTION TIME SUPPORT FOR ADAPTIVE SCIENTIFIC ALGORITHMS ON DISTRIBUTED MEMORY MACHINES	May 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Harry Berryman Joel Saltz Jeffrey Scroggs	90-41
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered Contractor Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to Concurrency, Practice and Experience

Final Report

16. Abstract

We consider optimizations that are required for efficient execution of code segments that consists of loops over distributed data structures. The PARTI execution time primitives are sedigned to carry out these optimizations and can be used to implement a wide range of scientific algorithms on distributed memory machines.

These primitives allow the user to control array mappings in a way that gives an appearance of shared memory. Computations can be based on a global index set. Primitives are used to carry out gather and scatter operations on distributed arrays. Communications patterns are derived at runtime, and the appropriate send and receive messages are automatically generated.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
unstructured mesh, adaptive, partial differential equations, distributed memory, gather, scatter, PARTI	59 - Mathematical and Computer Sciences (General) 61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	20	A03